Consistency Checking for the Evolution of Cardinality-based Feature Models

Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien, Goetz Botterweck

Consistency Checking for the Evolution of Cardinality-based Feature Models

Clément Quinton*, Andreas Pleuss+, Daniel Le Berre†, Laurence Duchien*, Goetz Botterweck+
*INRIA Lille - Nord Europe, LIFL UMR CNRS 8022, University Lille 1, France, {first.last}@inria.fr
+Lero – The Irish Software Engineering Research Center, University of Limerick, Ireland, {first.last}@lero.ie
†LIFL UMR CNRS 8022, Artois University, France, Daniel.Le-Berre@lifl.fr

ABSTRACT

Feature-models (FMs) are a widely used approach to specify the commonalities and variability in variable systems and software product lines. Various works have addressed edits to FMs for FM evolution and tool support to ensure consistency of FMs. An important extension to FMs are feature cardinalities and related constraints, as extensively used *e.g.*, when modeling variability of cloud computing environments. Since cardinality-based FMs pose additional complexity, additional support for evolution and consistency checking with respect to feature cardinalities would be desirable, but has not been addressed yet. In this paper, we discuss common cardinality-based FM edits and resulting inconsistencies based on experiences with FMs in cloud domain. We introduce tool-support for automated inconsistency detection and explanation based on an off-the-shelf solver. We demonstrate the feasibility of the approach by an empirical evaluation showing the performance of the tool.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Methodologies, Representation*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering, Reuse models*

General Terms

Design, Verification

Keywords

Cardinality, Feature Model, Edit, Consistency

1. INTRODUCTION

A key task in engineering *Software Product Lines* (SPLs) is specifying the variability between the products in a variability model [20]. One of the most common variability modeling approaches are *Feature Models* (FM) [25]. As SPLs are often a long-term investment they need to evolve to meet new requirements over many years [5]. This is reflected in the need to evolve FMs [10, 26, 19]. Various works deal with FMs and their evolution. For instance, the semantics of FMs has been formally defined enabling automated analysis [25]. Several approaches address FM analysis [2], *e.g.*, to detect inconsistencies that can arise during specification and evolution of FMs. Other work discusses edits (changes) on FMs during evolution, *e.g.*, with respect to the resulting set of products [28], the mappings to solution space [26] or FM consistency [10].

A frequently used extension of FMs are *cardinality-based* FMs which support *feature cardinalities* to specify how many instances (or clones) of a feature (and its subtree) can be included in a product configuration [6, 8]. For instance, in the domain of cloud computing, cardinality-based FMs can be used to configure cloud platforms and applications to be deployed on the cloud on different abstraction levels [22].

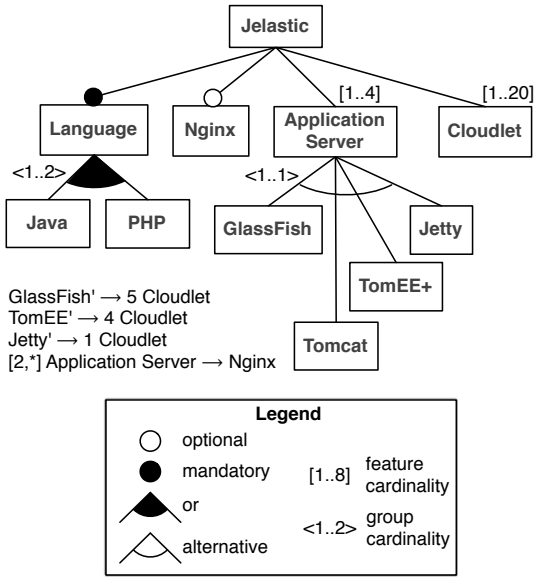


Figure 1: The Jelastic Feature Model (excerpt)

FIG. 1 shows a cardinality-based FM for the *Jelastic*[1] cloud environment. The FM enables to select a `Language`, an optional load balancer (`Nginx`), 1 to 4 instances of `ApplicationServer` and 1 to 20 instances of `Cloudlet`[2]. The available application server variants require different numbers of

[1] `http://jelastic.com/`

[2] A Cloudlet is the computation unit of Jelastic. It provides 128MB of RAM and 200MHz of CPU power.

SPLC '14, September 15 - 19 2014, Florence, Italy

http://dx.doi.org/10.1145/2648511.2648524.

`Cloudlet` instances to run properly and selecting 2 or more instances of `ApplicationServer` requires `Nginx`. Hence, the cardinality-based FM is complemented with constraints over the feature cardinalities [21], e.g., `GlassFish`' $\rightarrow$ 5 `Cloudlet` (each instance of `GlassFish` requires 5 `Cloudlet` instances). Details on the notation will be introduced in SEC. 2.

However, while feature cardinalities can add significant complexity to FMs and their constraints, support for inconsistency detection is still missing. In practice, FM configuration can be a complex task due to size and complexity of FMs [23]. It is hence strongly desirable to avoid additional complexity or extra effort (e.g., manually analyzing why a certain cardinality value cannot be selected) caused by an inconsistent FM. This paper addresses this need. We discuss FM edits during evolution with respect to feature cardinalities and present a formal approach to detect and explain inconsistencies in cardinality-based FMs. The approach has been implemented and evaluated with respect to its performance.

The paper is structured as follows: SEC. 2 introduces details of cardinality-based FMs followed by a motivating example highlighting the problem of inconsistencies that arise during evolution of such models. SEC. 3 presents a catalog of FM edits for cardinality-based evolution and their effects on FM consistency. SEC. 4 describes our approach to reason on this consistency and proposes an automated tool to detect cardinality-based inconsistencies. SEC. 5 presents an evaluation of the approach, SEC. 6 discusses related work, and SEC. 7 concludes the paper.

2. BACKGROUND AND MOTIVATION

This section introduces the details of cardinality-based FMs supported by our approach and illustrates the problems that might arise during their evolution by a motivating example.

2.1 Cardinality-based Feature Models

FMs were first introduced in 1990 [11] and various extensions have been proposed since then [25]. A basic FM defines available features and their dependencies in a tree hierarchy. The relationship between a parent feature and a child feature is constrained as *mandatory* (a child feature is required whenever its parent is selected) or *optional* (a child feature is optional). Alternatively, a feature can be part of a *feature group* where the group is constrained as *or-group* (at least one of the sub-features must be selected) or *alternative-group* (exactly one of the sub-features must be selected). Some approaches support constraining feature groups in a more generalized way by specifying a *group cardinality* <*m..n*> which defines the minimum and maximum number of sub-features to be selected. Or-groups can then be expressed by the group cardinality <1..n>, where n is the number of sub-features, and alternative-groups by <1..1>. In addition to the main hierarchy, *cross-tree constraints* can be used to describe dependencies between arbitrary features, *e.g.*, that selecting a feature *implies* the selection of another one or that two features mutually *exclude* each other. Some approaches support arbitrary Boolean constraints over features such as $A \rightarrow B \vee C$, where selecting feature A implies selecting feature B or feature C.

Cardinality-based feature models support in addition feature cardinalities [6, 8], first introduced as UML-like multiplicities [24]. A feature cardinality [*m..n*] can be assigned to any feature except a FM's root feature. It specifies how many instances[3] of a feature and its subtree can be included in a product configuration with m as lower bound and n as upper bound ($m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}$). In the same way as group cardinalities generalize constraints on feature groups, feature cardinalities generalize constraints on single features as they can also be used to express that a feature is mandatory ([1..1]) or optional ([0..1]). Note that for better readability we use the common notation for *or*, *alternative*, *mandatory* and *optional* in the diagrams (see legend in FIG. 1) but internally those are represented by cardinalities. As shown in the example in SEC. 1, there can be constraints over feature cardinalities. Considering a cardinality-based feature model $\mathcal{M}$, with $\mathcal{F}$ the non-empty set of features of $\mathcal{M}$, then the general form of such constraints with cardinalities is

$$\alpha\ F_{from} \rightarrow \beta\ F_{to} \text{ or } F_{from} \rightarrow n\ F_{to}$$

where

- $F_{from}, F_{to} \in \mathcal{F}$ with $F_{from} \neq F_{to}$;
- $n \in \mathbb{N}$;
- α and β define two intervals [i-j] and [m-n] with i, j, m, n $\in \mathbb{N}$ and i $\leq$ j, m $\leq$ n. α and β are the ranges over the set of required feature instances for F_{from} and F_{to} respectively;
- α, β and δ are optional.

Constraints over cardinalities can thus be specified, *e.g.*,:

$$[2,4]\ A \rightarrow [3,*]\ B \quad (1)$$

$$A' \rightarrow 2B \quad (2)$$

They constrain the number of instances of each feature in the configuration. For example, constraint (1) specifies that if there are at least two and at most four `A` instances in the configuration, then there must be also at least three `B` instances. We can also reason at the occurrence level. For example, constraint (2) defines that there must be twice more instances of `B` than `A` in the configuration, *i.e.*, for each (defined by the apostrophe next to the feature name) instance of `A`, there must be two instances of `B`. Such a constraint is satisfied off $\texttt{card}(B) \geq (n \times \texttt{card}(A))$, where $\texttt{card}: \mathcal{F} \rightarrow \mathbb{N}$ indicates the number of instances for a feature. More details about the semantics of such cardinality-based constraints can be found in our previous work [21].

2.2 Motivating Example

While the evolution of Boolean FMs as been widely studied in the literature [12, 1, 16, 14, 10, 28, 19, 26, 18, 17], little is known about cardinality-based FM evolution. For instance, cloud environments evolve over time, *e.g.*, when a new service support is available or the provided amount of resources has been extended. As a consequence, the corresponding cardinality-based FMs have to evolve.

FIG. 2 shows as example an evolution step on an extract of the Jelastic FM. The left-hand side shows an initial version that supports only `Jetty` as application server. The right-hand side shows the FM after the evolution where `GlassFish` has been added as an alternative application server together with the constraint C_2, as `GlassFish` is an heavy application server and requires 5 `Cloudlets` to work properly[4].

[3] also known as *clones* or *copies* in the literature. It seems to us that feature *instances* is more appropriate as they can differ in *e.g.*, a feature attribute value.

[4] `http://docs.jelastic.com/glassfish`

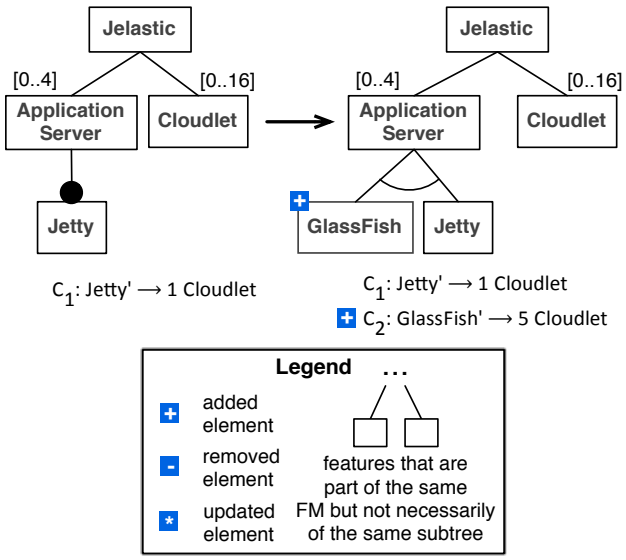


Figure 2: Evolution Example

However, this change makes the FM range inconsistent: according to the `ApplicationServer` feature cardinality, there can be up to 4 application server instances in a given configuration. However, configuring 4 `GlassFish` instances means that 20 `Cloudlet` instances are required, which is not allowed by the `Cloudlet` feature cardinality. This example demonstrates that manually evolving a cardinality-based FM is error-prone.

A feature cardinality is considered as *range inconsistent* if no product exists for some values of its range. This definition is analogous to the concept of *wrong cardinality* defined for group cardinalities in the existing literature [29, 2]. In the remainder of the paper, we discuss cardinality-based FM edits and the resulting range inconsistencies of feature cardinalities and introduce a formal approach to support detection and explanation of range inconsistencies in such FMs.

3. EVOLUTION OF CARDINALITY-BASED FEATURE MODELS

This section investigates basic edits to cardinality-based FMs with respect to consistency of feature cardinalities. We consider here atomic model edits only, i.e., to *add*, *remove* or *update* a model element where updating a model element means changing one of its properties. Here, relevant model elements are features and cross-tree constraints and relevant feature properties are name, feature cardinality, group cardinality, and location in the FM (i.e. the reference to the parent feature). This results in nine atomic FM edits discussed in this paper and listed in TABLE 1. An exclamation mark (!) indicates if the considered edit can lead to range inconsistencies while a check mark (✓) is used otherwise. In practice, knowing which edits do not lead to range inconsistencies is useful because it provides the possibility to save effort by not checking the FM after one of those edits has been performed. The table also describes if an edit is not applicable for the given model element.

	Add	Remove	Update	Move
Feature	✓	✓	✓	!
Feature Cardinality	NA	NA	!	NA
Group Cardinality	NA	NA	!	NA
Constraint	!	✓	!	NA

Table 1: Atomic edits

FM edits have already been discussed in existing work (see related work in SEC. 6) but, to the best of our knowledge, not with respect to feature cardinalities. We show in this section that five of the nine atomic edits can lead to range inconsistencies. More complex edits can be composed from atomic edits (e.g., *inserting* a feature into the feature tree hierarchy consists of adding a feature and moving a subtree to become its child) and can result in range inconsistencies if any of the involved atomic operators can lead to range inconsistencies. Note that range inconsistencies are always detected on feature cardinalities. If a change on a group cardinality leads to a range inconsistency for a feature cardinality, we consider that we have to fix the feature cardinality, not the group one.

Most of the scenarios described here arose during our previous work regarding cloud environment variability modeling [22]. However, we believe these scenarios may occur in any other domain and we therefore depict them in a general way, relying on the notation described in SEC. 2.2. (Please note that for mandatory features whose parent feature is omitted, we assume that they are part of every product, like mandatory children of the root feature).

Add Feature. The atomic edit *add* means adding a new feature as a leaf node to the model (as inserting a feature as non-leaf node requires to move an existing subtree to become its child). Adding a feature does not lead to range inconsistencies as it does neither influence any existing cardinalities nor any existing constraints.

Remove Feature. The atomic edit *remove* means to delete a feature (and its children) from the model. While this can lead to inconsistencies in general, such as dangling references in cross-tree constraints (in such a case, the involved constraint must be removed before removing the feature), it does not lead to range inconsistencies for the same reasons as the *add* edit.

Update Feature Name. Updating the name of a feature does not lead to range inconsistencies.

Move Feature. Moving features can occur during refactoring a model, when inserting new features (as non-leaf nodes), or removing non-leaf features. It can lead to range inconsistencies if there are hierarchies of cardinalities.

FIG. 3 depicts an example where feature `B` is moved within the FM to become a child of feature `C` (e.g., as part of inserting a new feature `C`). A range inconsistency can arise from the combined cardinalities of `C` and `B`. We rely on the local notion of feature instances already chosen by other authors [6, 13], that is, each instance of `C` provides up to two instances of `B`. As there can be up to three instances of `C`, there can be altogether up to six instances of `B`. The cross-tree constraint specifies that each instance of `B` requires two

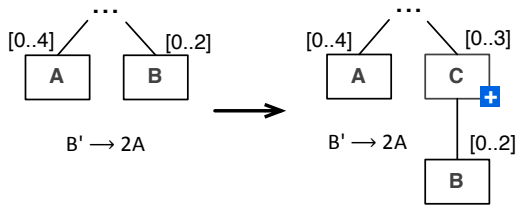
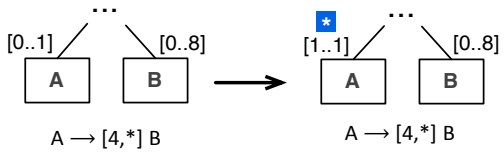


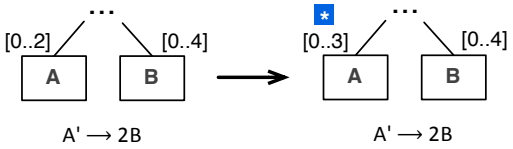
Figure 3: Moving a Feature

instances of A, i.e., up to 12 instances of A. But the feature cardinality of A allows only up to four instances which constitutes a range inconsistency.

Update Feature Cardinality. Each feature is given a cardinality, even mandatory and optional features, which are represented as cardinality [1..1] and [0..1] respectively. Thus, a cardinality cannot be removed or added but only updated which can result in range inconsistencies. Updating a feature cardinality occurs, for instance, when a cloud provider changes its service offer, *e.g.*, one can now run more than two database instances.



(a) Optional to mandatory



(b) Arbitrary cardinality

Figure 4: Updating Feature cardinality

Fig. 4 shows two examples for this scenario where the cardinality of feature A is updated. In Fig. 4a, A is an optional feature and is evolved to become a mandatory feature. The cross-tree constraint specifies that if A is selected there must be at least four instances of B. When A becomes mandatory there can never be less than four instances of B which is inconsistent with the feature cardinality of B.

In Fig. 4b, the upper bound changes from 2 to 3, but since the lower bound of the new cardinality is still 0, feature A remains an optional feature after the change. The cross-tree constraint specifies that each instance of A requires two instances of B. As the upper bound of B is still 4, it is not possible to configure three A instances which constitutes a range inconsistency.

Update Group Cardinality. A group cardinality may change when the system specifications evolve. For instance, the cloud environment might now support more different frameworks to be used in the same configuration. Updating a group cardinality can lead to range inconsistencies. In the scenario depicted by Fig. 5, one must now select at least two

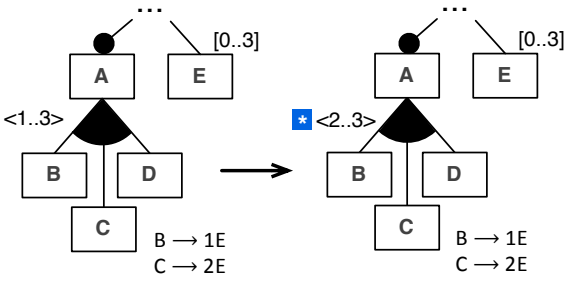


Figure 5: Updating a Group Cardinality

child features when A is configured, instead of one before the evolution. The constraints define the number of E instances to be configured when selecting B or C. After the evolution, either B or C must be selected (or both of them in the same configuration), which then requires at least one instance of E. However, this is inconsistent with the feature cardinality of E, which is defined as optional.

Add Cross-Tree Constraint. Adding a cross-tree constraint occurs, e.g., when a constraint is not part of the initial specification, but is added later based on experience with the system. It may also be added together with a new feature, *e.g.*, as depicted in Fig. 2. Both, cardinality-specific constraints and purely Boolean constraints can lead to range inconsistencies.

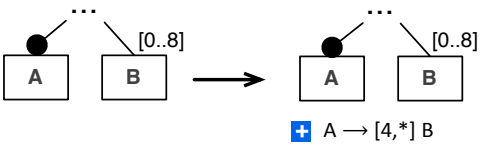


Figure 6: Adding a Constraint

In the example depicted by Fig. 6, a new constraint is added to specify that feature A requires at least four instances of B. As A is a mandatory feature there can never be less than four instances of B which is inconsistent with the feature cardinality of B.

Remove Cross-Tree Constraint. Removing a cross-tree constraint occurs when the constraint is not necessary anymore and cannot result in range inconsistencies. Given that the model is consistent before the evolution (*i.e.*, a feature is part of at least one product for each of its cardinality values), a range inconsistency can only arise if *(i)* at least one of the products is removed from the set of valid products or *(ii)* a new cardinality value is added to the set of cardinality values. However, removing a constraint can never restrict the set of products nor extend the set of cardinality values.

Update Cross-Tree Constraint. Updating a cross-tree constraint occurs, e.g., when an existing cloud service now provides less CPU power, thus requiring more instances to fit the same requirements than before. It can lead to range inconsistencies in the same way as adding cross-tree constraints. Fig. 7 shows an example where the cardinality range evolves. In this scenario, this edit leads to an inconsistency if A is selected, since feature A requires at least four instances of B, which is not possible.

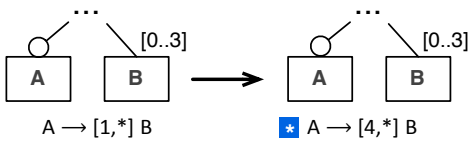


Figure 7: Updating a Constraint

This section has shown that among all atomic edits, five out of nine can result in range inconsistencies (see TABLE 1). Since more complex edits can be composed from these atomic edits, there is a high probability to get an inconsistent cardinality-based FM when evolving it manually. In the following section, we present a tool-supported approach to check the consistency of cardinality-based FMs.

4. INCONSISTENCY DETECTION

In the previous section, we described how edits applied on cardinality-based FMs can lead to inconsistent cardinalities. Some of those inconsistencies may seem obvious, *e.g.*, the one in the *add cross-tree constraint* scenario depicted by FIG. 6. However, the reader should keep in mind that features involved in these scenarios may be located in different subtrees of the FM, with tens of features and constraints in the FM, making those inconsistencies difficult to detect. Hence, there is need for automated tool support to detect and explain such inconsistencies.

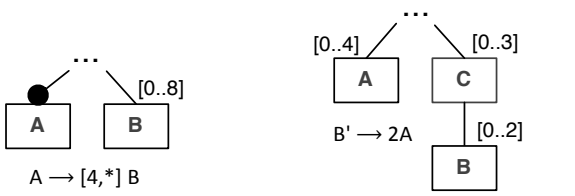
In this section, we first present a formal approach to detect inconsistencies based on a definition of consistency for cardinality-based FMs, and then propose a corresponding tool implementation based on an off-the-shelf solver.

4.1 Range Consistency Definition

To define consistency for cardinality-based FMs, we specify two closely related types of inconsistencies over feature cardinalities:

(1) *Local range inconsistency.* There is a local range inconsistency in the FM when there exists no product for one or several values defined by a cardinality range.

(2) *Global range inconsistency.* There is a global range inconsistency in the FM when a range inconsistency arises after taking into account the hierarchies of cardinalities.



(a) Local range inconsistency (b) Global range inconsistency

Figure 8: Cardinality Inconsistencies

Examples for local and global inconsistencies can be found in FIG. 4a and FIG. 3 after evolution has occurred, depicted here in FIG. 8 again for convenience. FIG. 8a shows a local inconsistency: as feature `A` is mandatory and due to the constraint, there must always be at least 4 instances of `B` while the feature cardinality of `B` specifies a lower bound of 0. This inconsistency occurs without having to take the cardinalities of parent features into account. FIG. 8b shows a global inconsistency: it only occurs due to the hierarchy of cardinalities. As there are up to 3 instances of `C`, each with up to 2 instances of `B`, there can be altogether up to `6` instances of `B`. Due to the cross-tree constraint this can require, in turn, up to 12 instances of `A` which conflicts with the feature cardinality of `A`. This inconsistency occurs only when taking the hierarchies of feature cardinalities (here `C` and `B`) into account.

Let $\mathcal{M} = (\mathcal{F}, \omega, \varphi)$ be a cardinality-based feature model with:

- $\mathcal{F}$ the non-empty set of features of $\mathcal{M}$;
- φ the set of constraints of $\mathcal{M}$;
- $\omega : \mathcal{F} \rightarrow \mathbb{N} \times \mathbb{N}$ indicates the cardinality of each feature. It will be denoted as a range $\omega(f) = [m, n]$.

In our context, a product may contain several instances of the same feature. It can either be represented as a multiset of feature names or as a set of pairs *(feature name, number of instances)*. We only require to retrieve the number of instances of a given feature for a given product. We denote $\mathcal{P}$ the set of all $\mathcal{M}$ products and $|f|_p$ the number of instances of the feature f in product p.

Definition 1. [LOCAL FM RANGE CONSISTENCY]
A cardinality-based feature model $\mathcal{M}$ is locally range consistent iff:

$$\forall f \in \mathcal{F}, \forall i \in \omega(f), \exists \{p \in \mathcal{P} \mid f \in p \ \wedge |f|_p = i\}$$

Thus, the FM depicted in FIG. 8a is not local range consistent, since there are values in ω(`B`) that are never used, *i.e.*, {0,1,2,3}. However, the FM in FIG. 8b is still local range consistent as there is a valid product configuration for each cardinality value of `B` (*e.g.*, if `C` is set to 1) and for each cardinality value of `C` (*e.g.*, if `B` is set to 0). Hence, relying on local range consistency is not enough to detect all cardinality inconsistencies in a FM, as it does not take into account the overall number of feature instances that can be configured due to hierarchies of ranges. We thus extend the previous definition to deal with this issue.

Definition 2. [GLOBAL FM RANGE CONSISTENCY]
A cardinality-based feature model $\mathcal{M}$ is globally range consistent iff:

$$\forall f \in \mathcal{F} \setminus \{root\}, g = \texttt{parent}(f), \forall j \in \omega(g),$$
$$\forall (x_1, ..., x_j) \in \omega(f) \times ... \times \omega(f),$$
$$\exists \{p \in \mathcal{P} \mid |g|_p = j \wedge \textstyle\bigwedge_{i=1}^{j} |f_i|_p = x_i\}$$

where $\texttt{parent}(f)$ is a function that indicates the parent feature of the f feature, *root* is the root feature of $\mathcal{M}$, and f_i denotes the instances of f attached to the *ith* instance of g in product p.

There is a relationship between local range and global range consistencies. The latter can be expressed using the

former on a normalized FM where a feature cardinality models the overall number of instances of that feature in the product. We translate a FM $\mathcal{M}$ into the normalized FM $\mathcal{M}'$ such that in $\mathcal{M}'$ the set of features remains the same than in $\mathcal{M}$ but feature cardinality ranges are updated to describe the number of feature instances. Such a translation is written

$$\mathcal{T} : \mathcal{M} = (\mathcal{F}, \omega, \varphi) \rightarrow \mathcal{M}' = (\mathcal{F}, \omega', \varphi)$$

with:

$$\omega'(f) = \omega(f) \textit{ iff } \texttt{parent}(f) = \varnothing$$
$$= \omega(f) \times \omega'(\texttt{parent}(f))$$

The relationship can be summarized as follows:

Theorem 1. [RELATIONSHIP BETWEEN CONSISTENCIES] *Checking the global range consistency of a* FM $\mathcal{M}$ *is equivalent to checking the local range consistency of* $\mathcal{T}(\mathcal{M})$.

For instance, regarding FIG. 8b, the idea is to replace the initial range [0..2] of `B` feature cardinality with [0..6], where $0 = 0 * 0$ and $6 = 2 * 3$. Checking the local range consistency of $\mathcal{M}'$ is, hence, performed regarding the range of feature instances ([0..6] is not the real cardinality of `B` but a range defining how many instances of `B` can be configured). In this example, the FM $\mathcal{M}'$ is not local consistent because there exists no product with more than two instances of `B`. We can show that checking global consistency in $\mathcal{M}$ is equivalent to checking local consistency in $\mathcal{M}'$.

We thus define a cardinality-based FM as a *range complete* FM if no local or global range inconsistency is detected. It is interesting to note that our notion of *range completeness* in FM is very close to the notion of *Global Inverse Consistency* (GIC) in the area of Constraint Satisfaction Problems as recently proposed by Bessiere *et al.* [4]. A CSP is *GIC* iff for every value in the domains of its variables, there exists a solution of the CSP with such value. A value in a domain for which there is no solution is called non-GIC. It is easy to show that detecting a local range inconsistency is detecting that there exists a non-GIC value in the CSP representing the FM. The benefit of relating our work to the one on *GIC* is that it allows us to reuse the tools to maintain *GIC* to detect such inconsistencies.

4.2 Automated Support

The approach described in this section has been incorporated into the SALOON platform, dedicated to editing and configuring cardinality-based FMs [21]. It relies on the *Eclipse Modeling Framework* (EMF) [27], a widely accepted framework to specify and implement metamodels. SALOON provides a FM metamodel, and FMs are defined compliant to this metamodel. One can thus create or edit FMs using the EMF editor or by directly editing the FM files, which are stored in XMI format, a XML-based format for model interchange. Once edited, FMs are first translated into a configuration problem using the BR4CP (Business Recommendation for Configurable Products[5]) textual format. Even though the BR4CP project supports two different formats, *i.e.*, CSP (XML format) and Aralia (textual) [9], we consider the latter as more intuitive since its syntax is closer to feature modeling constraints than the XML one, and makes the translation easier.

We thus developed an algorithm which takes as input a FM described using the XMI format and translates this FM into the BR4CP textual format. Two solvers implement *GIC* detection: one based on the Abscon CSP solver [4], and one based on SAT [3]. We use the latter because it provides us explanation support as well.

Listing 1: FM described in as configuration problem

```
1 #(1,1,[A.1]);
2 #(1,1,[B.0, B.1, B.2, B.3, B.4, B.5, B.6, B.7, B.8]);
3 (A.1 => B.4 | B.5 | B.6 | B.7 | B.8);
```

Once FMs are translated, the resulting text file is given as input to the solver. LISTING 1 shows the configuration problem resulting from the translation of the FM depicted in FIG. 8a. Lines 1 and 2 define the variables of the model, A and B, representing features `A` and `B` respectively. Values `1,1` just before the left square bracket specify how many values can be selected in the variable range, *i.e.*, *at least 1* and *at most 1*. Since `A` is mandatory, its unique possible value is 1, *i.e.*, [A.1], contrarily to feature `B` which holds as cardinality the range [0..8]. Line 3 describes the constraints $\texttt{A} \rightarrow$ [4,8]`B`. Thus, if the value of variable A is 1, then the value of variable B is either 4, 5, 6, 7 or 8.

```
Macintosh:cardFMs clement$ java -jar sat4j.jar fmEvo.txt
computing problem backbone...
rootPropagated: A=1
rootReduced: B=0 B=1 B=2 B=3
done in 0,012s with 5 SAT calls.

$> #explain -B=2
A.1=>B.4|B.5|B.6|B.7|B.8
$>
```

Figure 9: Inconsistency detection and explanation with Sat4j

Based on this input, the solver is used to *(i)* detect the inconsistencies, if any, and describe whether it's a local or a global range one and *(ii)* give feedback to the user to understand where and why such inconsistencies occur. FIG. 9 depicts the result once the inconsistencies detection algorithm is performed on the configuration problem described in LISTING 1 (given as argument in *fmEvo.txt*). As expected, the value of variable A is always 1 (*rootPropagated: A=1*). The next line is an inconsistency detection, meaning that the range value of variable B has been reduced, since values 0, 1, 2 and 3 are unreachable. The tool can then be used to understand where does such an inconsistency come from, using the *#explain* command. In this example, an explanation is asked to understand why variable B cannot be equal to 2. As a result, the constraint leading to the inconsistency is displayed: if variable A equals 1, then the value of B is greater than or equal to 4.

5. EVALUATION

In this section, we describe the implementation details and we report on some experiments we conducted to evaluate our

[5] `www.irit.fr/~Helene.Fargier/BR4CP/BR4CP.html`

approach. The intent of this evaluation is to assess the scalability of our approach when evolving large cardinality-based FMs and checking their consistency.

5.1 Setup and Methodology

The FMs described in SEC. 3 and the one used to illustrate the implementation details in SEC. 4.2 are very small examples, with less than 5 features. The aim of these experiments is to evaluate the performance of our approach when dealing with large FMs. However, we did not find large cardinality-based FMs in the literature. We thus implemented an algorithm to randomly generate such FMs.

For these experiments, we developed an algorithm that, given *nbFeatures* and *cardMax*, generates a random FM with *nbFeatures* features, whose cardinality is randomly assigned a range $\mathcal{R}$, with $\mathcal{R} \subseteq [0..cardMax]$. This algorithm works as follows. It creates *nbFeatures* features, then randomly builds the tree hierarchy. More precisely, while there exists remaining features, it randomly selects a given amount of these features, assigns them a tree level value and increments this value, which gives the tree depth. For instance, given $nbFeatures = 10$, a random tree hierarchy with 4 levels is $\{\{f1,f2,f3\},\{f4\},\{f5,f6\},\{f7,f8,f9,f10\}\}$. Then, for each feature of a given level, it randomly assigns a given amount of child features, if possible. In the previous example, if $f4$ has already been assigned as a child of $f1$, then $f2$ and $f3$ have no child feature. For features having more than one child, the algorithm determines if the relationship is a basic parent-child relationship, an alternative or an exclusive group (33% probability each). The algorithm also generates cross-tree constraints, where two features are selected randomly, with one constraint for every 10 features as proposed by Thüm *et al.* [28]. Either Boolean or cardinality-based constraints are generated (50% probability each). For the latter, the ranges of required feature instances are built to fit within the feature cardinality so that the generated FM is consistent. In our experiments, we only consider non-void FMs, that is, FMs with at least one valid configuration, so void FMs (due to the generated random constraints) are discarded. We performed our experiments on a MacBook Pro with a 2.6 GHz Intel Core i7 processor and 8 GB of DDR3 RAM.

To measure the performance of our approach to detect range inconsistencies, we also generate FM edits. In particular, regarding the scenarios described in SEC. 3, we implemented the following operations: (1) move a feature, (2) add a cross-tree constraint, (3) update a feature cardinality and (4) update a cross-tree constraint. Operation (1) picks at random an existing feature, and assigns it as a child or a parent of another randomly selected feature. In (2), the algorithm generates a random constraint, while a feature cardinality is randomly changed by operation (3). Finally, operation (4) changes a Boolean constraint into a cardinality-based constraint or updates the range of the existing cardinality-based constraint. For each generated FM, we generate one or several edits leading, on purpose, to range inconsistency. For each edit, we thus know what kind of inconsistency will result. For instance, we update a cross-tree constraint and check the consistency of the targeted feature, which becomes local range inconsistent.

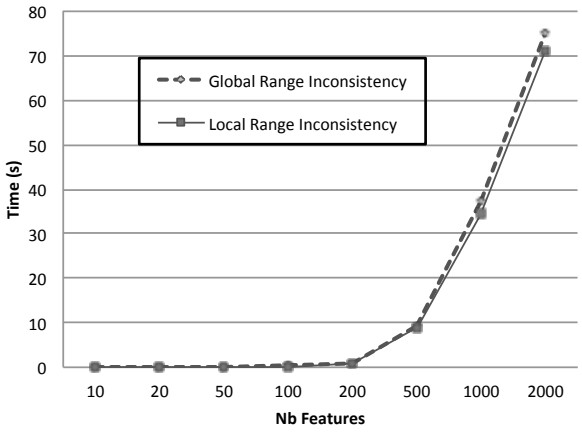


Figure 10: Detecting an inconsistency in FMs with maximum upper bound cardinality set randomly from 1 to 10.

We then evaluated how our approach performs when handling cardinalities, in particular regarding the potential combinatorial explosion. Indeed, the number of combinations that the solver has to examine grows significantly when using cardinalities. To reduce this issue and improve our inconsistency detection mechanism, we adapt the algorithm described in DEFINITION 2. In our experiments, to check the global range consistency of a feature f, the translation algorithm does not consider all ancestors of f but only one of them, namely the nearest ancestor that has a cardinality upper bound greater than one (if any). This improvement is used only for these experiments, as the approach described in SEC. 3 handles any cardinality-based FMs. We argue that this adaptation is fair since, in all cardinality-based FMs we found in the literature, including ours, we never found any FM whose cardinality upper bound is greater than one for more than two features hierarchically linked, *e.g.*, A:[0..3], B:[1..5] and C:[0..4] with A parent of B and B parent of C.

5.2 Experimental Results

For the first experiment, we measure the computation time required to find an inconsistency with *(i)* one random edit leading to a local or global range inconsistency, *(ii)* a fixed value for *cardMax* and *(iii)* an increasing number of features *nbFeatures*, thus varying the size of the FM. We set the value of *cardMax* to 10. We argue that this value is a fair value to evaluate our approach. First, as explained above, we never found any FM whose cardinality upper bound is greater than one for more than two features hierarchically linked. Second, in the case we found a feature with a high *cardMax* number, its parent feature was either an optional or mandatory feature with *cardMax* set to 1. Our algorithm does not take these findings into consideration and generates FMs with a random cardinality for each feature, *i.e.*, FMs whose combinatorial complexity grows significantly with FM size.

In this experiment, we vary the size of the FM from 10 to 2000 features which is, once again, more significant than the cardinality-based FMs we found, including ours. Generating random FMs leads to important differences regarding computation time to detect inconsistency for the same FM size. This is due to the randomly generated tree hierarchy and

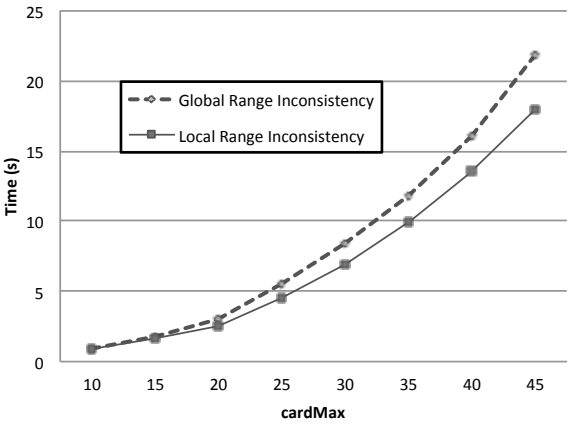


Figure 11: Detecting inconsistency while varying the maximum upper bound cardinality (FM size = 200).

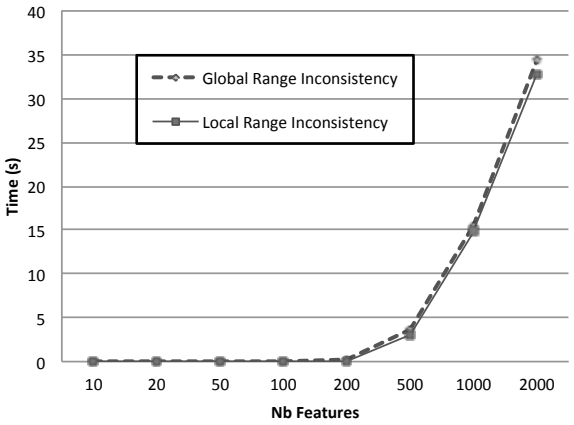


Figure 12: Detecting an inconsistency in FMs whose one feature out of ten has its cardinality upper bound greater than one.

constraints as well as the random edits performed. To deal with this issue, we performed 200 generation runs for each *nbFeatures* value and computed the average time. As shown in FIG. 10, our detection algorithms perform the same way to detect both local and global range inconsistencies, even if detecting a global range inconsistency generates a small overhead compared to detecting a local one, *e.g.*, 2,5 seconds for 1000 features. This overhead results from the additional variables and constraints the solver has to handle, due to the way we translate the FM into the textual configuration problem format as described in SEC. 4.

In general, cardinality-based FMs require the solver to handle a higher number of variables than, *e.g.*, Boolean FMs, which explains the difference to Boolean FMs (*e.g.*, about 1 second to find an edit for a FM with 2000 features [28]). Given a feature f with a cardinality range [0..8], then 9 variables are required to reason on this feature, *i.e.*, one per cardinality value. For instance, for cardinality-based FMs with 2000 features and 200 constraints, the solver has for an average model to reason over 15670 variables and 11910 constraints to detect local range inconsistencies and over 16120 variables and 12240 constraints to detect global range inconsistencies. Overall, for cardinality-based FMs with less than 200 features, the computation time is less than 1 second. This time then increases significantly for larger FMs, with an average time of 9 seconds for 500 features, 36 seconds for 1000 features and up to 73 seconds for 2000 features.

For the second experiment, we measure the computation time required to find an inconsistency with *(i)* one random edit leading to a local or global range inconsistency, *(ii)* an increasing value for *cardMax* and *(iii)* a fixed number of features, with *nbFeatures* = 200. We consider this value is fair for this experiment, as this is larger than the biggest cardinality-based FM we found. To generate FMs with features whose cardinality is the one expected, we modified our algorithm so that a feature is given a cardinality [m..n], with m ∈ {0,1} and n = *cardMax*. The aim of this experiment is to evaluate how our approach performs in presence of several features with a high cardinality. We thus vary *cardMax* from 10 to 45 and compare the computation time required to detect either a local or global range inconsistency. We performed 500 generation runs for each *cardMax* value and computed the average time. FIG. 11 depicts the results of this experiment. As expected, the time required to detect an inconsistency increases as the value of *cardMax* does, from less than one second for *cardMax* = 10 to more than 18 seconds for *cardMax* = 45 (18 and 21.8 seconds in average to detect a local or a global range inconsistency respectively). As in the previous experiment, detecting a global range inconsistency generates an overhead compared to detect a local one, *e.g.*, one second for *cardMax* = 25 or two seconds for *cardMax* = 35.

For the third experiment, we measure the computation time required to find inconsistencies with *(i) nbEdits* random edits leading to local or global range inconsistencies, with *nbEdits* ∈ [1..*nbFeatures*/20], *(ii)* a random value for *cardMax* with *cardMax* ∈ [1..30] and *(iii)* an increasing number of features *nbFeatures*, thus varying the size of the FM. Our generation algorithm is slightly evolved to generate random FMs where one out of ten features has its cardinality upper bound set to *cardMax*. Other features are either optional or mandatory features as usually found in Boolean FMs. We then generate random edits and compute the time required to find an inconsistency, if any. The aim of this experiment is to evaluate how our approach performs when checking cardinality-based FMs whose structure is as close as possible to the one we met in the literature. Moreover, this is not limited to one edit, but several random ones, as it occurs in classic feature oriented development, *e.g.*, staged configuration [7]. FIG. 12 depicts the results of this experiments, where the time is an average computation time over 200 generation runs. The resulting trend matches the one depicted in FIG. 10, but in this setting our approach requires less than half of the time to detect inconsistencies, *e.g.*, 15 seconds for 1000 features in this experiment in comparison to 36 seconds for the same number of features in the first experiment. We would expect such a result, as there are fewer features with a high cardinality upper bound value. The time required to detect an inconsistency remains however quite significant. This is due to the value of *cardMax* (200 features with a cardinality upper bound set up to 30 for a FM with 2000 features) and to the random number of edits, *e.g.*, up to 50 for a FM with 1000 features.

Overall, our approach is well-suited to detect inconsistencies in cardinality-based FMs whose size is lower than 500 features (about 9 seconds in the worst case, 3 seconds otherwise). For larger FMs, the number of features as well as the cardinality upper bound value have an important impact on the computation time. However, although we did not define a threshold for this experiment, we can fairly argue that these results are acceptable, as we did not find in the literature such large cardinality-based FMs.

5.3 Threats to Validity

There are several concerns in our approach that may form threats to validity. Regarding our evaluation, we tested our approach on randomly generated cardinality-based FMs. Even though it seems fair, we did implement the generation algorithm and thus expect the FMs to conform a certain structure, which is based on our previous work in this domain. This might not be representative of the usage of cardinality-based FMs, in particular in the industrial domain. Moreover, the algorithm does not generate completely random FMs, since they are consistent after being generated. The generation process is thus guided to fit once again a certain structure. One could also argue that our evaluation is not complete as it does not take global range consistency for the whole list of ancestors into account, but only for one of them. However, such a result would only serve as comparison, as we did not find any cardinality-based FM matching this pattern.

6. RELATED WORK

This section discusses existing work on FM edits and on reasoning over FMs.

Feature Model edits. Several works discuss the way FMs evolve. Alves *et al.* investigated issues that need to be addressed when refactoring SPLs [1]. They provide a catalog of edits for refactoring FMs, *e.g.*, *add new alternative* or *replace mandatory feature*. Lotufo *et al.* study the evolution of a real world variability model, the Linux kernel one [12]. They describe what operations are performed to evolve this variability model, and empirically assess their findings. Pleuss *et al.* rely on a model-driven support to handle FM evolutions [19]. The FM is reified into a high-level FM, EvoFM, clustered into fragments which are added or removed regarding the expected evolution. Guo *et al.* also propose an approach to check the consistency of evolving FMs [10], and analyze the semantics of FM changes using ontologies. Some authors also focus on changes at different level, *i.e.*, in the problem space, the mapping space and/or the solution space. Neves *et al.* describe safe evolution templates for product lines [14]. These templates preserve the behavior of existing products and can be applied to both feature model and artifacts level. Seidl *et al.* describe four different evolutions at feature level (duplicate, split, insert and remove) [26]. They provide remapping operators when co-evolving SPLs in the FM and the related feature mappings. Passos *et al.* also focus their work on the evolution of the variability model together with the source code [17, 18]. They extend Lotufo's work, using as example the Linux kernel variability model, and describe a catalog of evolution patterns for the co-evolution of the variability model and the related artifacts. These approaches do not consider cardinality-based FMs.

Czarnecki *et al.* proposed some evolution scenarios that can be applied to cardinality-based FMs [6]. However, these edits are only considered as specialization steps, *i.e.*, where the set of configurations after the evolution is a subset of the set of configurations before the evolution. For example, regarding a feature cardinality, they only consider reducing the range of instances that may be configured. Moreover, cardinality-based constraints are not taken into consideration in their approach. In contrast, this paper addresses arbitrary edits and takes constraints over the feature cardinalities into account.

Reasoning. Several works deal with reasoning and explanations on evolving FMs. Thüm *et al.* [28] provide a tool that analyzes changes performed on a FM and classifies them into (1) refactoring, not changing the set of valid products, (2) generalization, only adding products, (3) specialization, reducing the set of products, or (4) arbitrary changes otherwise. A systematic overview on previous work on feature model analysis is provided in [2]. For instance, various work supports detection of anomalies in an FM, such as "dead features" (a feature can never be selected) or "false optional features" (a feature is specified as optional but must be included into all products, *e.g.*, due to cross-tree constraints). While the majority of existing work addresses Boolean FMs, some work supports detecting such anomalies also in cardinality-based FMs, *e.g.*, based on propositional logic [8] or a knowledge-base [15]. The survey also considers inconsistent cardinalities ("wrong cardinality") but only with respect to group cardinalities and, moreover, none of the surveyed work supports detecting them.

To the best of our knowledge, none of the existing approaches supports detecting inconsistent FMs with respect to feature cardinalities and constraints over them, as presented in this paper.

7. CONCLUSION

Feature modeling is a well-known approach for variability modeling in SPLs. As SPLs often represent a long-term investment and need to meet new requirements, managing their evolution has become a key factor. Thus, evolving the underlying FM becomes necessary. While previous work has addressed evolution and consistency checking for Boolean FMs, feature cardinalities and their constraints have not been considered yet. This paper addressed this gap by discussing atomic FMs edits with respect to feature cardinality and describing how such edits can lead to inconsistency. An approach is proposed to automatically detect such inconsistencies and support the FM designer by explaining *where*, *why* and *what kind* of inconsistency arose when editing the model. All the described approach has been implemented in a prototypical tool support, and integrated to the SALOON framework, dedicated to the management of cardinality-based FMs. Our empirical evaluation showed that our approach checks the consistency of large cardinality-based FMs (2000 features) in a reasonable time compared to their related combinatorial complexity, due to our modeling optimization.

Future work could implement our approach in a more integrated visual tool support, *e.g.*, by propagating the explanations given by the solver to the FM graphical editor and highlighting model elements leading to the inconsistency. One could also consider other kind of inconsistencies, *e.g.*, *conditionally* local range inconsistency, that is, a feature cardi-

nality is inconsistent under certain conditions, or extend our approach to features models extended with attributes.

Acknowledgments

The authors thank Daniel Romero for its valuable comments. This work is partially supported by the EU FP7 PaaSage project, BR4CP ANR project and by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (`www.lero.ie`).

8. REFERENCES

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring Product Lines. In *GPCE'06*, pages 201–210. ACM, 2006.

[2] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Inf. Syst.*, 35(6):615–636, 2010.

[3] D. L. Berre and A. Parrain. The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[4] C. Bessiere, H. Fargier, and C. Lecoutre. Global Inverse Consistency for Interactive Constraint Satisfaction. In *CP 2013*, pages 159–174. Springer, 2013.

[5] G. Botterweck and A. Pleuss. Evolution of software product lines. In *Evolving Software Systems*, pages 265–295. Springer, 2014.

[6] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[7] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[8] K. Czarnecki and C. H. P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *International Workshop on Software Factories at OOPSLA'05*. ACM, 2005.

[9] Y. Dutuit and A. Rauzy. Exact and Truncated Computations of Prime Implicants of Coherent and non-Coherent Fault Trees within Aralia. *Reliability Engineering and System Safety*, 58(2):127–144, 1997.

[10] J. Guo, Y. Wang, P. Trinidad, and D. Benavides. Consistency Maintenance for Evolving Feature Models. *Expert Syst. Appl.*, 39(5):4987–4998, 2012.

[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, The Software Engineering Institute, 1990.

[12] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the Linux Kernel Variability Model. In *SPLC'10*, pages 136–150. Springer, 2010.

[13] R. Michel, A. Classen, A. Hubaux, and Q. Boucher. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In *VaMoS'11*, pages 82–89. ACM, 2011.

[14] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the Safe Evolution of Software Product Lines. In *GPCE'11*, pages 33–42. ACM, 2011.

[15] A. Osman, S. Phon-Amnuaisuk, and C. K. Ho. Knowledge Based Method to Validate Feature Models. In *SPLC'08 Workshops*, pages 217–225. Lero, 2008.

[16] P. Paskevicius, R. Damasevicius, and V. Štuikys. Change Impact Analysis of Feature Models. In *ICIST 2012*, pages 108–122. Springer, 2012.

[17] L. Passos, K. Czarnecki, and A. Wąsowski. Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case. In *FOSD'12*, pages 62–69. ACM, 2012.

[18] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, and P. Borba. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *SPLC'13*, pages 91–100. ACM, 2013.

[19] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski. Model-driven Support for Product Line Evolution on Feature Level. *J. Syst. Softw.*, 85(10):2261–2274, 2012.

[20] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[21] C. Quinton, D. Romero, and L. Duchien. Cardinality-based Feature Models with Constraints: A Pragmatic Approach. In *SPLC'13*, pages 162–166. ACM, 2013.

[22] C. Quinton, D. Romero, and L. Duchien. Automated Selection and Configuration of Cloud Environments Using Software Product Lines Principles. In *IEEE CLOUD*, 2014.

[23] R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Inf. Softw. Technol.*, 52(3):324–346, 2010.

[24] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *IDPT 2002*, 2002.

[25] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *RE'06*, pages 136–145. IEEE Computer Society, 2006.

[26] C. Seidl, F. Heidenreich, and U. Aßmann. Co-evolution of Models and Feature Mapping in Software Product Lines. In *SPLC'12*, pages 76–85. ACM, 2012.

[27] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2009.

[28] T. Thüm, D. Batory, and C. Kastner. Reasoning About Edits to Feature Models. In *ICSE'09*, pages 254–264. IEEE Computer Society, 2009.

[29] P. Trinidad and A. Ruiz–Cortés. Abductive Reasoning and Automated Analysis of Feature Models: How are they connected? In *VaMos'09*, page 145–153. Universität Duisburg-Essen, 2009.